CREATING RESOURCE AGENTS FOR COLLABORATIVE ENGINEERING ENVIRONMENT (CEE) RESEARCH USING THE COMMON OBJECT REQUEST BROKER ARCHITECTURE (CORBA) FRAMEWORK

March 2001

MAJ JOHN M. EMMERT, LUIS CONCHA, LT KEITH PEDERSEN, CAPT DIANE STARKEY

AIR FORCE RESEARCH LABORATORY
AIR FORCE MATERIAL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7334

REPORT DOCUMENTATION PAGE

1. REPORT DATE (DD-MM-YYYY)	2. REPORT TYPE	3. DATES COVERED (FROM - TO)
27-03-2001	Technical Report	xx-xx-2001 to xx-xx-2001

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Creating Resource Agents for Collaborative Engineering	5b. GRANT NUMBER
Unclassified	5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)	5d. PROJECT NUMBER
Emmert, John M. ; Concha, Luis M. ; Pedersen, Keith E. ; Starkey, Diane L. ;	5e. TASK NUMBER
	5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME AND ADDRESS	8. PERFORMING ORGANIZATION REPORT NUMBER
Air Force Research Laboratory Air Force Material Command Wright-Patterson AFB , OH 45433-7334	

9. SPONSORING/MONITORING AGENCY NAME AND ADDRESS	10. SPONSOR/MONITOR'S ACRONYM(S)
,	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

A
PUBLIC RELEASE

,

13. SUPPLEMENTARY NOTES

14. ABSTRACT

In this report we describe the setup of a collaborative engineering environment (CEE) for performing research on the performance of distributed systems. We provide the basis for implementing a multi-tiered host network that is capable of operating across several different computing platforms without modification to system software. The CEE allows multiple processes to be launched on remote systems taking advantage of distributed processing capabilities. The object broker within the environment provides a platform to test and analyze ?smart? network software. Smart network software includes but is not limited to algorithms and methods to determine which available processor is the best choice for performing a particular task or operation.

15. SUBJECT TERMS
CORBA; CEE; Distributed Computing; Collaborative Engineering; Resource Agents

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
			Public Release	31	Fenster, Lynn lfenster@dtic.mil
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**19b. TELEPHONE NUMBER** **International Area Code** **Area Code Telephone Number** 703 767-9007 **DSN** 427-9007
Unclassified	Unclassified	Unclassified			

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*
27-03-2001

2. REPORT TYPE
technical

3. DATES COVERED *(From - To)*

4. TITLE AND SUBTITLE
Creating Resource Agents for Collaborative Engineering

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Emmert, John M; Author
Concha, Luis M; Author
Pedersen, Keith E; Author
Starkey, Diane L; Author

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT
A Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
In this report we describe the setup of a collaborative engineering environment (CEE) for performing research on the performance of distributed systems. We provide the basis for implementing a multi-tiered host network that is capable of operating across several different computing platforms without modification to system software. The CEE allows multiple processes to be launched on remote systems taking advantage of distributed processing capabilities. The object broker within the environment provides a platform to test and analyze "smart" network software. Smart network software includes but is not limited to algorithms and methods to determine which available processor is the best choice for performing a particular task or operation.

15. SUBJECT TERMS
CORBA; CEE; Distributed Computing; Collaborative Engineering; Resource Agents

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**	Unclassified Unlimited	30	John M. Emmert
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED			**19b. TELEPHONE NUMBER** *(include area code)* 704-687-4323

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

1. INTRODUCTION

1.1. Most software applications are designed for a specific hardware platform and operating system. However, often an application is required on an unsupported platform. One way to address this problem is to *port* the application's code and compile it for the unsupported platform. This typically requires many tedious, error prone man-hours of work, and after the code has been compiled for the unsupported platform, it often results in less than optimal performance. In order to avoid this overhead and poor performance, a better solution is to execute the application on the platform for which it was designed and *interface* to the application from the unsupported platform. This solution requires interoperability among various platforms.

1.2. Timely, accurate information is vital to Air Force war fighting capability. Often, required information makes use of servers or data available at remote locations accessible through a host network. This requires interoperability among various platforms and operating systems.

1.3. To address interoperability issues and intercommunication among applications written for different operating systems executing on various hardware platforms, the Object Management Group[i] (OMG) has created the Common Object Request Broker Architecture[ii] (CORBA) framework. The CORBA framework was introduced in 1991 to allow applications to communicate with each other no matter where they are located. They did this by defining a standard protocol by which applications can communicate through the use of an interface description language (IDL) interface. The IDL is a language-independent specification to provide programmers the option of choosing the most appropriate operating system and platform for creating and executing a specific application. They also allow the integration of various applications through the use of a network. There are actually many other Agent types available[1]; however, we chose the CORBA system based on its wide usage and it is distributed freely.

1.4. The IDL interface allows us to create and execute software agents independent of any operating system or execution platform[2]. For example a C^{++} program written for a unix based workstation could be executed from a Pentium based PC running MSDOS without the need to recompile the code for the unix workstation. Software agents communicate together using an *agent communication language*. The agent provides a message based interface between objects[3]. Several definitions have been applied to *agent*[4]:

1.4.1. MuBot Agent: The term *agent* is used to represent two orthogonal concepts. First, the agents ability for autonomous execution. Second, the ability for the agent to perform domain oriented reasoning[5].

[i] Information on the OMG is available at http:\\www.omg.org
[ii] Information on CORBA is available at http:\\www.corba.org

1.4.2. AIMA Agent: Anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors[6].

1.4.3. Maes Agent: Computational systems that inhabit some complex dynamic environment, sense and act autonomously in this environment, and by doing so realize a set of goals or tasks for which they are designed[7]

1.4.4. KidSim Agent: Persistent software entity dedicated to a specific purpose[8].

1.4.5. Hayes-Roth Agent: Agents continuously perform three functions. First, perception of dynamic conditions in the environment. Second, action to affect conditions in the environment. Third, reasoning to interpret perceptions, solve problems, draw inferences, and determine actions[9].

1.4.6. IBM Agent: Software entities that carry out some set of operations on behalf of user or another program with some degree of independence or autonomy[10].

1.4.7. Wooldridge-p; Jennings Agent: A hardware or software-based computer that enjoys the following properties: autonomy, social ability, reactivity, and pro-activeness[11].

1.4.8. SodaBot Agent: Programs that engage in dialogs and negotiate and coordinate transfer of information[12].

1.4.9. Brustoloni Agent: Systems capable of autonomous, purposeful action in the real world[13].

In summary, agents are capable of operating autonomously and yet maintaining the ability to communicate socially.

1.5. For this report, we will define the *client* as the process requesting work, and a *server* as the agent performing the work. Agents have the capability of both receiving instructions and returning information. To reiterate, clients and servers are independent (they may be co-located or physically separated, and they need not execute on the same platform), but they retain the ability to communicate through an ORB interface described by an IDL file. For example, the server can be written in C++ while the client is written in Java. Additionally, the client and server can share the same hard drive, or they can be networked i.e. through the World Wide Web.

1.6. The objective of this project is to research and become familiar with information technologies as applied to the Collaborative Enterprise Environment (CEE), thereby enabling an in-house capability to develop new resource agents for CEE. This phase I report documents initial lessons learned in the installation, and execution of CORBA applications using the CORBA compliant software available free from ORBacus[iii]. The downloadable CORBA software for both C++ and Java is available at http://www.ooc.com/ob/download. The report is broken down into the following sections. First we describe the location and installation of the CORBA software for unix, linux, and Windows based operating systems. Then we describe the use of the IDL for creating interfaces between C++ and/or Java programs. Next we describe how to write clients and servers using both C++ and Java code. We include both inter- and intra- network examples. Finally, in an appendix, we provide makefiles and scripts necessary for each step of the process. Throughout this report we will provide complete working examples for each step.

2. LOCATING AND INSTALLING THE CORBA SOFTWARE

2.1. In order to run applications (clients and servers) under the ORBacus[14] CORBA software, the files for the target platform must be downloaded. These files can be found at the ORBacus web page: http://www.ooc.com/ob/download.html. For installing the Java version, the file JOB-3.1.3.tar.gz should be downloaded (or JOB-3.1.3.zip). For installing the C++ version, the file OB-3.1.3.tar.gz should be downloaded. For instructions on ORBacus, a manual is available in PDF form. The manual for the ORBacus software can be downloaded from OB-3.1.3.pdf.gz. To use ORBacus for Java without having to compile ORBacus for C++, you can download the pre-compiled IDL-to-Java translator. For Linux this translator is jidl-3.1.3-linux.tar.gz, for Solaris this translator is jidl-3.1.3-sloaris.tar.gz, and for Windows 95/98/NT the translator is jidl-3.1.3-win32.tar.gz. Several JAR files are also available for Java by downloading JOB-3.1.3.jars.tar.gz (these will be needed for creating Java classes). Two additional sources of information on CORBA are the book *Understanding CORBA* by Otte et. al.[15] and *The Common Object Request Broker: Architecture and Specification, Rev 2.2*[16]

2.2. The installation of the programs is straight forward once the installation files are unzipped. To unzip the files, type gunzip *filename*.tar.g then untar the files using tar cvf *filename*.tar (where *filename* is the name of the installation package). If the file is from a PC and in the form *filename*.tgz just type gunzip *filename*.tgz. Then to install the specific package, follow the platform specific directions that come with the package. NOTE: The environment settings or variables a user needs to set depend upon the local configuration of the hardware platform and should be specified in the instructions for installing the package. Please see your system administrator if additional help is needed.

[iii] Information on the ORBacus CORBA compliant software is available at http:\\www.ooc.com

3. CREATING IDL FILES

3.1. Interface description language (IDL) files define the interface between the *server* and the *client*. The complexity of these files can range from low to high (depending on the size of the interface, the number and type of error handlers, etc…). We provide two, low complexity example interfaces to get started. We describe the components of each, and we describe how to compile the components in order to get the sub-components necessary to allow the clients and servers to function correctly.

3.2. The first interface is simple. There is no interaction between the client and server other than the client causes the server to execute. This interface can be used in a local network or it can be accessed remotely through URL connections. The following interface definition is stored in the file `Hello.idl`.

```
// IDL
module MOD_HELLO
{
  // interface definitions
  interface Int_hello
  {
    void funct_hello();
  };
};
```

In CORBA IDL, comments are denoted using "`//`". A `module` is similar to a package or class. It is optional, and ties the interfaces and functions together. In this example, the module name is `MOD_HELLO`. The module name is user defined, but must remain consistent throughout the applications. Next, we have the interfaces. There can be several in the same module. The `interface` is the gateway between servers and clients. It defines the *handshaking* or communication between the servers and clients. In this example, the interface name is `Int_hello`. The interface name is also user defined. Inside the interface, there are one or more objects. In this example there is one user defined object: `funct_hello`. This object is also user defined. In this case, the object neither receives nor returns any information. The client can use this object to ask the server to perform some task. Now that the basic IDL is written, we can create the files required by the server and client to use this IDL.

3.2.1. First, we will create the files needed to create servers and clients using C++. To do this we use the ORBacus IDL translator "`idl`". We execute the following from the command line:

```
idl Hello.idl
```

The result of this command is the creation of several files: `Hello.cpp`, `Hello.h`, `Hello_skel.cpp`, and `Hello_skel.h`. These files are

required when compiling. The client will require `Hello.cpp` and `Hello.h`. The server will require `Hello.cpp, Hello.h, Hello_skel.cpp` and `Hello_skel.h`. In addition, the server will also require the files `Hello_impl.cpp` and `Hello_impl.h`. The `Hello_impl` files contain the actual implementation of the function or object the server will implement. The directions for writing simple servers, clients, and implementations are found in subsequent sections.

3.2.2. The files and objects required to implement servers and clients in Java is similar to that of C++. To create these files we use the ORBacus Java command line translator "`jidl`" in the following command line:

```
jidl Hello.idl
```

This will create a sub or class directory `MOD_HELLO` that contains several Java based files required for compiling (creating the Java classes) and executing the Java code.

3.3. The second interface is slightly more complicated than the first. This interface allows the client to provide information to the server (in this example the client provides an integer value). The server then acts on the information and provides data back to the client for further processing (in this example the server provides an integer back to the client). This example is designed to show communication between the client and the server. Like the previous example, this IDL interface can be used for clients and servers that share the same local network or for clients and servers on different networks. The example below is stored in file `Counter.idl`

```
// IDL
module COUNTER_MODULE
{
  // variable definitions
  typedef long N;  // length of count

  // interface definitions
  interface Count_interface
  {
    long count_funct(in long N);
  };
};
```

This IDL file is similar to the first example, except a variable is passed through the interface from client to server, and an integer value is returned from the server.

3.3.1. Similar to the previous example, to create the files for a C++ client and/or server, from the command line type

```
idl Counter.idl
```

This will create four files for the C++ implementation. Creation of the server and client using C++ will be described in sections 4 & 6 respectively.

3.3.2. To create the files necessary for Java implementation execute

```
jidl Counter.idl
```

This creates the class directory structure for the Java implementation of the server and client. The Java server and client will be discussed in sections 5 & 7 respectively.

3.4. This section describes the basic approach for implementing IDL descriptions for two types of interfaces. One interface requires no information back from the server. It is designed to cause the server to execute some specific command or commands. The second is an example of an interface that passes data between the client and server. In following sections, we will describe the C++ and Java implementations that accompany the IDL descriptions.

4. CREATING C++ RESOURCE AGENTS (SERVERS)

4.1. In this section we describe the process to develop simple resource agents or *servers* using the C++ high level programming language. We provide complete examples and instructions for compiling and executing the server. The first example is executed when requested by a client. It prints a predefined message to the screen, and it returns no information to the client. The second example takes as input a positive integer, *N*, value from the client. It then counts from 1 to *N*, while at the same time adding the integers from 1 to *N*. After counting is complete, it returns the sum of the integers from 1 to *N*.

4.2. The code, compiling instructions, and execution instructions for the first example are described below.

4.2.1. First we have the code for the server implementation. It is stored in file `Server.cpp`

```
// C++
#include <OB/CORBA.h>
#include <Hello_impl.h>
#include <fstream.h>
int main(int argc,char* argv[],char*[])
{
  CORBA_ORB_var orb = CORBA_ORB_init(argc,argv);
  CORBA_BOA_var boa = orb -> BOA_init(argc,argv);

  // Program calls module “MOD_HELLO”, interface
  // “Int_hello” and creates a new object
```

```
  // implementation from the Hello implementation
  MOD_HELLO_Int_hello_var p = new Hello_impl;

  CORBA_String_var s = orb ->object_to_string(p);
  const char* refFile = "Hello.ref";
  ofstream out (refFile);
  out << s << endl;
  out.close();

  // Server waits until called
  boa -> impl_is_ready(CORBA_ImplementationDef::_nil());
}
```

4.2.2. The server needs an implementation object. The header for the implementation body is found in the file `Hello_impl.h` and shown below:

```
//  C++
#include <Hello_skel.h>
class Hello_impl : public MOD_HELLO_Int_hello_skel
{
 public:
  // function defined for use by the server
  virtual void funct_hello();
};
```

4.2.3. The code for the server implementation is stored in the file `Hello_impl.cpp` and shown below.

```
// C++
#include <OB/CORBA.h>
#include <Hello_impl.h>
void
Hello_impl::funct_hello()
{
  cout << "Hello from AFRL/IFSD!" << endl;
}
```

This file, when executed will print a simple message from AFRL.

4.2.4. To compile the code for the server, the make file (found in section 10.1) can be executed with the `make` utility available from most C++ compilers. For unix the following command line will work:

```
make –f  hello_server_make_file
```

4.2.5. To execute the server, from the command line execute the following command:

```
Server
```

This will cause the server to execute until it is terminated with a kill command or `^c`. Usually, the server will go into a wait state, where it will stay until it is instantiated by the client.

4.2.6. In this section we have shown how to write, compile, and execute simple servers for use with an IDL. A key point to note is that a server written in C++ is compatible with a client written in Java and vice versa. In the next part of this section we will give an example that can be executed from a remote URL.

4.3. The code, compiling instructions, and execution instructions for the second example are shown and described below.

`4.3.1.` First we have the code for the server implementation. It is stored in `Counter_server.cpp`.

```
// C++
#include <OB/CORBA.h>
#include <OB/Util.h>
#include <OB/CosNaming.h>
#include <Counter_impl.h>
#include <fstream.h>
#include <stdio.h>
int main(int argc,char* argv[],char*[])
{
  try
  {
    // Create ORB
    CORBA_ORB_var orb = CORBA_ORB_init(argc,argv);
    // Create BOA
    CORBA_BOA_var boa = orb -> BOA_init(argc,argv);
    // Create implementation
    COUNTER_MODULE_Count_interface_var p = new counter_impl;
    // Get naming service
    CORBA_Object_var obj;
    try
    {
      obj = orb -> resolve_initial_references("NameService");
    }
    catch(const CORBA_ORB::InvalidName&)
    {
      cerr << argv[0] << ": can't resolve `NameService'" <<
              endl;
      return 1;
    }
    if(CORBA_is_nil(obj))
    {
      cerr << argv[0] << ": `NameService' is a nil object
              reference" << endl;
      return 1;
    }
    CosNaming_NamingContext_var count =
              CosNaming_NamingContext::_narrow(obj);
    if(CORBA_is_nil(count))
    {
      cerr << argv[0]
              << ": `NameService' is not a NamingContext”
```

```
            << " object reference"  << endl;
    return 1;
  }
  try
  {
    // Bind names with the Naming Service
    CosNaming_Name pName;
    pName.length(1);
    pName[0].id = CORBA_string_dup("p");
    pName[0].kind = CORBA_string_dup("");
    count -> bind(pName, p);
    boa -> impl_is_ready(CORBA_ImplementationDef::_nil());
    // Unregister name with the Naming Service
    count -> unbind(pName);
  }
  catch(const CosNaming_NamingContext::NotFound& ex)
  {
    cerr << argv[0] << ": Got a `NotFound' exception (";
    switch(ex.why)
    {
       case CosNaming_NamingContext::missing_node:
           cerr << "missing node";
           break;
       case CosNaming_NamingContext::not_context:
            cerr << "not context";
            break;
       case CosNaming_NamingContext::not_object:
            cerr << "not object";
            break;
    }
    cerr << ")" << endl;
    return 1;
  }
  catch(const CosNaming_NamingContext::CannotProceed&)
  {
    cerr << argv[0] << ": Got a `CannotProceed exception"
            << endl;
    return 1;
  }
  catch(const CosNaming_NamingContext::InvalidName&)
  {
    cerr << argv[0] << ": Got an `InvalidName' exception"
            << endl;
    return 1;
  }
  catch(const CosNaming_NamingContext::AlreadyBound&)
  {
    cerr << argv[0] << ": Got an `AlreadyBond' exception"
             << endl;
    return 1;
  }
  catch(const CosNaming_NamingContext::NotEmpty&)
  {
    cerr << argv[0] << ": Got a `NotEmpty' exception"
            << endl;
    return 1;
  }
}
catch(CORBA_SystemException& ex)
{
  OBPrintException(ex);
  return 1;
}
```

```
  return 0;
}
```

4.3.2. Next we have the code for the counter implementation header file. It contains the function definitions used to implement the counter function in the class `Counter_impl`, and it is stored in file `Counter_impl.h`

```
//  C++
#include <Counter_skel.h>
class Counter_impl : public
COUNTER_MODULE_Count_interface_skel
{
public:
  virtual long count_funct(long);
};
```

4.3.3. Next we have the function implementations for the functions in the class `Counter_impl`. These are stored in the file `Counter_impl.cpp`

```
// C++
#include <OB/CORBA.h>
#include <Counter_impl.h>
long Counter_impl::count_funct(long N)
{
  long sum = 0;
  long i = 0;
  while(++i <= N){
    cout << i << "." << endl;
    sum = sum + i;
  } // while
  return sum;
}
```

4.3.4. To compile the counter object, we use the makefile, `counter_server_make_file`, found in section 10.2. From the command line type:

```
make –f counter_server_make_file
```

4.3.5. To execute we first need the name server running. To start the name server, we need an IP address, a port number, and a mode of operation. For our, example we use IP = 10.13.7.7, port 1700 (see system administrator for other available port numbers), and `DefaultNamingContext` as the mode. More information on these can be found in the CORBA specification. To start the name server, execute the following line:

```
nameserv –i –OAport 1700 > Counter.ref &
```

Then to start the actual server, execute the following command line:

```
Counter_server -ORBservice NameService
            iiop://10.13.7.7:1700/DefaultNamingContext
```

Or to start the name server then the counter server both, execute the script `start_counter_server` shown in section 10.5 of the appendix.

4.3.6. In this section we described the method for executing C++ servers remotely using a URL.

5. CREATING JAVA RESOURCE AGENTS

5.1. Simple Java example

5.1.1. Code for Java server found in file `Server.java`

```
// Java
package MOD_HELLO;
public class Server
{
 public static void main(String args[])
 {
  java.util.Properties props = System.getProperties();
  props.put("org.omg.CORBA.ORBClass", "com.ooc.CORBA.ORB");
  props.put("org.omg.CORBA.ORBSingletonClass",
      "com.ooc.CORBA.ORBSingleton");
  System.setProperties(props);
  org.omg.CORBA.ORB orb =
  org.omg.CORBA.ORB.init(args, props);
  org.omg.CORBA.BOA boa =
              ((com.ooc.CORBA.ORB)orb).BOA_init(args, props);
  Int_hello_impl p = new Int_hello_impl();
  try
  {
   String ref = orb.object_to_string(p);
   String refFile = "Hello.ref";
   java.io.PrintWriter out = new java.io.PrintWriter(new
         java.io.FileOutputStream(refFile));
   out.println(ref);
   out.flush();
  }
  catch(java.io.IOException ex)
  {
    System.err.println("Can't write to `" + ex.getMessage() +
       "'");
    System.exit(1);
  }
  boa.impl_is_ready(null);
 }
}
```

5.1.2. Code for Java implementation found in file `Int_hello_impl.java`

```
// Java
```

```
package MOD_HELLO;
public class Int_hello_impl extends _Int_helloImplBase
{
 public void funct_hello()
 {
  System.out.println("Hello Analisa!");
 }
}
```

5.1.3. To compile the Java code, make sure all code is in the sub-directory `MOD_HELLO`. Then type the following command line:

```
javac ./MOD_HELLO/*.java
```

5.1.4. To execute the Hello Server using Java, type the following command line:

```
java MOD_HELLO.Server
```

5.2. Counter Java example

5.2.1. Code for `Counter_server.java`

```
package COUNTER_MODULE;
import org.omg.CORBA.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import java.io.*;
import java.util.*;
import COUNTER_MODULE.Counter_impl.*;

public class Counter_server
{
  public static void main(String args[])
  {
    Properties props = System.getProperties();
    props.put
       ("org.omg.CORBA.ORBClass", "com.ooc.CORBA.ORB");
    props.put
       ("org.omg.CORBA.ORBSingletonClass","com.ooc.CORBA.
       ORBSingleton");
    System.setProperties(props);
    try
    {
      //
      // Create ORB
      //
      ORB orb = ORB.init(args, props);
      //
      // Parse command line arguments
      //
      String namingFile = null;
      int argc = 0;
      while(argc < args.length)
      {
        if(args[argc].equals("-f"))
        {
```

```
    if(argc + 1 < args.length)
    {
      argc++;
      namingFile = args[argc];
    }
    else
      usage();
  }
  else if(args[argc].equals("-h") ||
            args[argc].equals("--help"))
  usage();
  argc++;
}
//
// Create BOA
//
BOA boa =((com.ooc.CORBA.ORB)orb).BOA_init(args,props);
//
// Create some implementations
//
Count_interface p = new Counter_impl();
//
// Get naming servie
//
org.omg.CORBA.Object obj = null;
if(namingFile != null)
{
  try
  {
    FileReader r = new FileReader(namingFile);
    BufferedReader in = new BufferedReader(r);
    String ref = in.readLine();
    r.close();
    obj = orb.string_to_object(ref);
  }
  catch(IOException e)
  {
    e.printStackTrace();
    System.exit(1);
  }
}
else
{
  try
  {
    obj = orb.resolve_initial_references("NameServic);
  }
  catch(org.omg.CORBA.ORBPackage.InvalidName ex)
  {
    System.out.println("Can't resolve `NameService'");
    System.exit(1);
  }
}
if(obj == null)
{
  System.out.println ("`NameService' is a nil object
      reference");
  System.exit(1);
}
NamingContext count =
      NamingContextHelper.narrow(obj);
if(count == null)
{
```

```
      System.out.println("`NameService' is not " +
             "a NamingContext object reference");
      System.exit(1);
    }
    try
    {
      //
      // Bind names with the Naming Service
      //
      NameComponent[] pName = new NameComponent[1];
      pName[0] = new NameComponent();
      pName[0].id = "p";
      pName[0].kind = "";
      count.bind(pName, p);
      //
      // Run implementation
      //
      boa.impl_is_ready(null);
      //
      // Unregister names with the Naming Service
      //
      count.unbind(pName);
    }
    catch(NotFound ex)
    {
      System.err.print("Got a `NotFound' exception (");
      switch(ex.why.value())
      {
        case NotFoundReason._missing_node:
             System.err.print("missing node");
             break;
        case NotFoundReason._not_context:
             System.err.print("not context");
             break;
        case NotFoundReason._not_object:
             System.err.print("not object");
             break;
      }
      System.err.println(")");
      ex.printStackTrace();
      System.exit(1);
    }
    catch(CannotProceed ex)
    {
      System.err.println ("Got a `CannotProceed'
             exception");
      ex.printStackTrace();
      System.exit(1);
    }
    catch(InvalidName ex)
    {
      System.err.println ("Got an `InvalidName' exception");
      ex.printStackTrace();
      System.exit(1);
    }
    catch(AlreadyBound ex)
    {
      System.err.println ("Got an `AlreadyBound'
              exception");
      ex.printStackTrace();
      System.exit(1);
    }
  }
```

```
    catch(SystemException ex)
    {
      ex.printStackTrace();
      System.exit(1);
    }
    System.exit(0);
  }
  private static void
  usage()
  {
    System.out.println(
       "Usage: java naming.Server [options]\n\n" +
       "Options:\n" +
       "-f FILE Read the NamingService IOR from file
              FILE.\n" +
       "-h, --help  Display this help message.");
    System.exit(1);
  }
}
```

5.2.2. Code for `Counter_impl.java`

```
package COUNTER_MODULE;
public class Counter_impl extends
           _Count_interfaceImplBase
{
public int count_funct(int n)
  {
    int sum = 0;
    int i = 0;
    while(++i <= n){
      sum = sum + i;
      System.out.println(i + ".");
    }
    return sum;
  }
}
```

5.2.3. To compile the Java code, make sure all Counter Java code is in the sub-directory `COUNTER_MODULE`. Create the script file from section 10.9 and call it `script_java` .Then type:

```
script_java
```

5.2.4. Similar to the C++ Counter Server, the name server must be executing for the server to run properly. Follow the directions in the script `start_counter_java_server` found in section 10.6 of the appendix.

6. CREATING C++ CLIENTS

6.1. Simple C++ example client.

6.1.1. This is the code for a simple C++ client, named `Client.cpp`.

```
// C++
#include <OB/CORBA.h>
#include <Hello.h>
#include <fstream.h>
int
main(int argc,char* argv[], char*[])
{
  CORBA_ORB_var orb = CORBA_ORB_init(argc,argv);
  const char* refFile = "Hello.ref";
  ifstream in(refFile);
  char s[1000];
  in >> s;
  CORBA_Object_var obj = orb -> string_to_object(s);
  MOD_HELLO_Int_hello_var hello =
                    MOD_HELLO_Int_hello::_narrow(obj);
  hello -> funct_hello();
}
```

6.1.2. To compile the code for the simple C++ client, use the makefile, `hello_client_make_file` found in section 10.3 of the appendix, and type:

```
make –f hello_client_make_file
```

6.1.3. To execute the simple hello client type `Client` from the command line. With the server already running, starting the client should cause the server to print a simple message to its machine.

6.2. Counter C++ example

6.2.1. Here is the code for the Counter Client found in the file `Counter_client.cpp`

```
// C++
#include <stdio.h>
#include <stdlib.h>
#include <OB/CORBA.h>
#include <OB/Util.h>
#include <OB/CosNaming.h>
// include files generated by CORBA idl
#include <Counter.h>
#include <fstream.h>
#include "functs.h"
int main(int argc,char* argv[], char*[])
{
  try
  {
     // Create ORB
     CORBA_ORB_var orb = CORBA_ORB_init(argc,argv);
     // Get naming service
     CORBA_Object_var obj;
     try
     {
       obj = orb -> get_inet_object
          ("10.13.7.7",1700,"DefaultNamingContext");
    }
```

```
catch(const CORBA_ORB::InvalidName&)
{
  cerr << argv[0]
           << ": can't resolve `NameService'" << endl;
  return 1;
}
if(CORBA_is_nil(obj))
{
  cerr << argv[0]
           << ": `NameService' is a nil object reference"
           << endl;
  return 1;
}
CosNaming_NamingContext_var count =
CosNaming_NamingContext::_narrow(obj);
if(CORBA_is_nil(count))
{
  cerr << argv[0]
          << ": `NameService' is not a NamingContext
          object reference"
          << endl;
  return 1;
}
try
{
  // Resolve names with the naming service
  CosNaming_Name pName;
  pName.length(1);
  pName[0].id = CORBA_string_dup("p");
  pName[0].kind = CORBA_string_dup("");
  CORBA_Object_var pObj = count -> resolve(pName);
  COUNTER_MODULE_Count_interface_var p =
  COUNTER_MODULE_Count_interface::_narrow(pObj);
  assert(!CORBA_is_nil(p));
  cout << "Resolved `p'" << endl;
  functs FUNCTS;
  long N;
  N = (long)FUNCTS.string2int(argv[1]);
  cout << "     The true sum = "
           << ((N+1)*N)/2 << endl;
  cout << "The simulated sum = "
           << p -> count_funct(N) << endl;
}
catch(const CosNaming_NamingContext::NotFound& ex)
{
  cerr << argv[0] << ": Got a `NotFound' exception (";
  switch(ex.why)
  {
    case CosNaming_NamingContext::missing_node:
        cerr << "missing node";
        break;
    case CosNaming_NamingContext::not_context:
        cerr << "not context";
        break;
    case CosNaming_NamingContext::not_object:
        cerr << "not object";
        break;
  }
  cerr << ")" << endl;
  return 1;
}
catch(const CosNaming_NamingContext::CannotProceed&)
{
```

```
        cerr << argv[0] << ": Got a `CannotProceed'
               exception" << endl;
        return 1;
      }
      catch(const CosNaming_NamingContext::InvalidName&)
      {
        cerr << argv[0] << ": Got an `InvalidName'
        exception" << endl;
        return 1;
      }
      catch(const CosNaming_NamingContext::AlreadyBound&)
      {
        cerr << argv[0] << ": Got an `AlreadyBound'
        exception" << endl;
        return 1;
      }
      catch(const CosNaming_NamingContext::NotEmpty&)
      {
        cerr << argv[0] << ": Got a `NotEmpty' exception"
             << endl;
        return 1;
      }
    }
    catch(CORBA_SystemException& ex)
    {
      OBPrintException(ex);
      return 1;
    }
    return 0;
  }
```

6.2.2. Since this file uses helper functions noted by the statement `#include "functs.h", the dependent library files functs.h & functs.cpp` found in section 10.7 & 10.8 of the appendix need to be available for compilation.

6.2.3. To compile the Counter Client, use the make file `counter_client_make_file` found in section 10.4 of the appendix and type:

```
make –f counter_client_make_file
```

6.2.4. The code for finding the name server is found inside the client. To execute with name server inside type:

```
Counter_client N
```

where *N* is a positive integer. The server should count to *N* and return the sum. The client should print out the sum.

7. CREATING JAVA CLIENTS

7.1. Simple Java Example

7.1.1. Code for file `Client.java`:

```
// Java
package MOD_HELLO;
public class Client
{
  public static void main(String args[])
  {
    java.util.Properties props = System.getProperties();
    props.put("org.omg.CORBA.ORBClass",
           "com.ooc.CORBA.ORB");
    props.put("org.omg.CORBA.ORBSingletonClass",
           "com.ooc.CORBA.ORBSingleton");
    System.setProperties(props);
    org.omg.CORBA.ORB orb =
    org.omg.CORBA.ORB.init(args, props);
    String ref = null;
    try
    {
      String refFile = "Hello.ref";
      java.io.BufferedReader in = new
      java.io.BufferedReader(
      new java.io.FileReader(refFile));
      ref = in.readLine();
    }
    catch(java.io.IOException ex)
    {
      System.err.println("Can't read from `" +
      ex.getMessage() + "'");
      System.exit(1);
    }
    org.omg.CORBA.Object obj = orb.string_to_object(ref);
    Int_hello p = Int_helloHelper.narrow(obj);
    p.funct_hello();
  }
}
```

7.1.2. To compile the Java code, make sure all `*.java` files are in the directory `MOD_HELLO`. Then type the following command line:

```
javac ./MOD_HELLO/*.java
```

7.1.3. To execute the Java client,

```
java MOD_HELLO.Client
```

This should cause the Server to print a message based on what is in the particular server implementation.

8. SUMMARY

8.1. It is not necessary to create the files described above in any particular order. Often times a program is written in C++. The IDL allows access to this program from other high level languages and other platforms than the one for which the code was written. CORBA is also useful for legacy systems. Interfaces can be written to existing code, thereby eliminating the tedious, error prone task of porting code. IDL over the internet, addresses two issues related to systems level

engineering. First, it helps eliminate redundancy. By allowing engineers and engineering teams to make use of remote resources, we eliminate the need to reproduce work that has already been accomplished elsewhere. Second, IDL enhances communication. IDL creates the framework to remotely execute many programs and applications. This allows leaders to sit in a central location and make use to resources and expertise scattered throughout the world.

9. FUTURE WORK

9.1. Create resource agents for Air Force programs. One good candidate is the CENTS program that is PIed by Adaptive Inc. They are interfacing different databases and different types of objects. Since this is the primary goal of CORBA systems, their program would be an ideal match for demonstrating the capability. (It also provides the opportunity to enhance the CEE system and commercialize results of SBIR efforts.)

9.2. Write intelligent ORBs that have the ability to choose one server over another to accomplish a particular task. Often times one server may be the ideal choice for executing a server application; however, there are times when the identified server may be overloaded with other processes. In this case the ORB should make the decision to transfer the request to another server for execution. This option has not been explored in this report.

9.3. Link the Clients, Servers, and ORBs to the CEE environment and/or an html based browser like Netscape or Explorer.

10. APPENDIX

10.1. This is the makefile: `hello_server_make_file`

```
#hello_server_make_file
# Compiler variables
CC = CC
#CC = g++

CFLAGS = -fast -pta -I/usr1/local/include -I.

# Create list of libraries to link in
LIBS = -lOB -lsocket -lnsl

# Program name
PROGRAM = ./Server

# source files
CC_SOURCES = Hello.cpp Hello_skel.cpp Hello_impl.cpp Server.cpp

# Create list of .o objects
OBJECTS = $(CC_SOURCES:.cpp=.o)

# Override the built-in compile
%.o : %.cpp
$(COMPILE.c) $<
```

```
# Top level target for building the executable
all :  $(PROGRAM)

# Define how to link all the stuff
$(PROGRAM) : $(OBJECTS)
$(LINK.c) $(OBJECTS) -o $(PROGRAM) $(LIBS)

###############################################################################
#####
# Clean up only the object files
clean:
\rm  *~ *.o core $(PROGRAM)
```

10.2. This is the makefile: `counter_server_make_file`

```
# Compiler variables
CC = CC
CFLAGS = -fast -pta -I/usr1/local/include -I.

# Create list of libraries to link in
LIBS = -lCosNaming -lOB -lsocket -lnsl

# Program name
PROGRAM = ./Counter_server

# source files
CC_SOURCES = Counter.cpp Counter_skel.cpp // Counter_impl.cpp
Counter_server.cpp


# Create list of .o objects
OBJECTS = $(CC_SOURCES:.cpp=.o)

# Override the built-in compile
%.o : %.cpp
$(COMPILE.c) $<

# Top level target for building the executable
all :  $(PROGRAM)

# Define how to link all the stuff
$(PROGRAM) : $(OBJECTS)
$(LINK.c) $(OBJECTS) -o $(PROGRAM) $(LIBS)

###############################################################################
#####
# Clean up only the object files
clean:
\rm  *~ *.o core $(PROGRAM)
```

10.3. This is the makefile: `hello_client_make_file`

```
#makeclient
# Compiler variables
CC = CC

CFLAGS = -fast -pta -I/usr1/local/include -I.

# Create list of libraries to link in
LIBS = -lOB -lsocket -lnsl
```

```
# Program name
PROGRAM = ./Client

# source files
CC_SOURCES = Hello.cpp Client.cpp

# Create list of .o objects
OBJECTS = $(CC_SOURCES:.cpp=.o)

# Override the built-in compile
%.o : %.cpp
$(COMPILE.c) $<

# Top level target for building the executable
all :  $(PROGRAM)

# Define how to link all the stuff
$(PROGRAM) : $(OBJECTS)
$(LINK.c) $(OBJECTS) -o $(PROGRAM) $(LIBS)

#############################################################################
#####
# Clean up only the object files
clean:
\rm  *~ *.o core $(PROGRAM)
```

10.4. This is the makefile: `counter_client_make_file`

```
# Compiler variables
CC = CC

CFLAGS = -fast -pta -I/usr1/local/include -I.

# Create list of libraries to link in
LIBS = -lCosNaming -lOB -lsocket -lnsl

# Program name
PROGRAM = ./Counter_client

# source files
CC_SOURCES = functs.cpp Counter.cpp Counter_client.cpp

# Create list of .o objects
OBJECTS = $(CC_SOURCES:.cpp=.o)

# Override the built-in compile
%.o : %.cpp
$(COMPILE.c) $<

# Top level target for building the executable
all :  $(PROGRAM)

# Define how to link all the stuff
$(PROGRAM) : $(OBJECTS)
$(LINK.c) $(OBJECTS) -o $(PROGRAM) $(LIBS)

#############################################################################
#####
# Clean up only the object files
clean:
\rm  *~ *.o core $(PROGRAM)
```

10.5. This is the script `start_counter_server`:

```
#!/bin/sh
#
echo
echo "*****************************"
echo "*  Welcome to the Server     *"
echo "*****************************"
sleep 2
nsid=0
srvid=0
deactivate()
{
    if test $nsid -ne 0
    then
      echo "killing "
      echo "$nsid"
      kill -9 $nsid
    fi

    if test $srvid -ne 0
    then
      kill -9 $srvid
    fi

    exit
}
server="nameserv${exe}"
ref=Counter.ref
rm -f $ref
$server -i -OAport 1700 > $ref &
nsid=$!
trap deactivate 1 2 3 4 5 6 7 8 10 12 13 14 15
echo "$ref"
echo "$nsid"
sleep 3
Counter_server -ORBservice NameService
iiop://10.13.7.7:1700/DefaultNamingContext
srvid=$!
deactivate
```

10.6. This is the script `start_counter_java_server`:

```
#!/bin/sh
#
echo
echo "*****************************"
echo "*  Welcome to the Server     *"
echo "*****************************"
sleep 2
nsid=0
srvid=0
deactivate()
{
    if test $nsid -ne 0
    then
      echo "killing "
      echo "$nsid"
      kill -9 $nsid
    fi
```

```
    if test $srvid -ne 0
    then
        kill -9 $srvid
    fi

    exit
}
CLASSPATH=/usr1/OB-3.1.3/ob/lib/OB.jar:$CLASSPATH
CLASSPATH=/usr1/OB-3.1.3/ob/lib/OBNaming.jar:$CLASSPATH
export CLASSPATH

server="nameserv${exe}"
ref=Counter.ref
rm -f $ref
$server -i -OAport 1700 > $ref &
nsid=$!
trap deactivate 1 2 3 4 5 6 7 8 10 12 13 14 15
echo "$ref"
echo "$nsid"
sleep 300
java  COUNTER_MODULE.Counter_server -ORBconfig
java_configuration.txt
deactivate
```

The shell script above uses a configuration text file with the IP address, port address, and mode inside of it. The name of the script is `java_configuration.txt` and its contents are shown below:

```
# Concurrency models
ooc.orb.conc_model=threaded
ooc.boa.conc_model=thread_pool
ooc.boa.thread_pool=5

#Initial services
ooc.service.NameService=iiop://10.13.7.7:1700/DefaultNamingContext
```

10.7. This is the library file `functs.h.` This module contains helper functions used by other classes.

```
//functs.h
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#include <math.h>
#include <stddef.h>
#include <ctype.h>

class functs {
 private:

 public:
  functs(){}; // constructor
  ~functs(){free_memory();}; // destructor
  void free_memory(){}; // frees dynamic memory

  int string2int(char*); // converts a string to an integer
  void int2string(int,char*); // converts and integer to a string
  int random_funct(int); // returns a random integer
```

```
  void upper2lower(char*,char*); // converts upper case to lower
  float max(float,float); // returns maximum float
  float min(float,float); // returns minimum float
  double perms(int,int); // calculates number of permutations
}; // functs
```

10.8. This file, named `functs.cpp`, defines `functs.h` above.

```
// functs.cpp
#ifndef _functs_def
  #define _functs_def
  #include "functs.h"
#endif

/*********************************************************/
/*                                                       */
/* perms                                                 */
/*                                                       */
/*********************************************************/
//
// Returns the number of permutations.
//
double functs::perms(int I,int J){
  double answer;
  int i;

  if((J > I) || (I < 0) || (J <
                          0)){printf("ERROR!\n");exit(1);}
  answer = 1.0;
  i = 1;
  while(++i <= I){
    answer *= (double)i;
  } // while

  J = I - J;
  i = 1;
  while(++i <= J){
    answer /= (double)i;
  } // while

  return answer;
}; // perms

/*********************************************************/
/*                                                       */
/* int2string                                            */
/*                                                       */
/*********************************************************/
//
// Converts and integer to a string.
//
void functs::int2string(int n,char* number){
  int i;int k;int j;
  int num[4];
  i = 4;while(--i > -1){j=(int)pow(10.0,(double)i);k =
             n/j;num[i] = k;n-=(k*j);}
  j = 0;i = 4;
  number[0]='0';number[1]='\0';
  while(--i > -1){
    if(num[i]>0){
      ++i;
      number[i] = '\0';
```

```
      while(--i > -1){
        if(num[i]==0){number[j]='0';}
        else if(num[i]==1){number[j]='1';}
        else if(num[i]==2){number[j]='2';}
        else if(num[i]==3){number[j]='3';}
        else if(num[i]==4){number[j]='4';}
        else if(num[i]==5){number[j]='5';}
        else if(num[i]==6){number[j]='6';}
        else if(num[i]==7){number[j]='7';}
        else if(num[i]==8){number[j]='8';}
        else if(num[i]==9){number[j]='9';}
        else {printf("Error: int2string\n");exit(1);}
        j++;
      } // while
    } // if
  } // while
}; // int2string

/*********************************************************/
/*                                                       */
/* upper2lower                                           */
/*                                                       */
/*********************************************************/
//
// Converts upper to lower case.
//
void functs::upper2lower(char* lower ,char* upper){
  int n = strlen(upper);
  int i=-1;while(++i<n){
    if((upper[i]>='A')&&(upper[i]<='Z')){lower[i]=(char)((int)
         upper[i]|(int)32);}
    else{lower[i]=upper[i];}
  }
  lower[n]='\0';
}; // upper2lower

/*********************************************************/
/*                                                       */
/* min                                                   */
/*                                                       */
/*********************************************************/
//
// Returns the minimum of two floats.
//
float functs::min(float a,float b){
  float answer;
  if(a < b){answer = a;}
  else{answer = b;}
  return answer;
}; // min

/*********************************************************/
/*                                                       */
/* max                                                   */
/*                                                       */
/*********************************************************/
//
// Returns the maximum of two floats.
//
float functs::max(float a,float b){
  float answer;
  if(a > b){answer = a;}
  else{answer = b;}
```

```
  return answer;
}; // max

/*******************************************************/
/*                                                     */
/* random_funct                                        */
/*                                                     */
/*******************************************************/
//
// This function returns a random number.
//
// Assumptions:  -- Integers are represented as 2's comp numbers
//               -- Each memory location is 8 bits (1 byte)
//               eg: 16 bits => a range of values from
//                   - ( 2 ^ 15 ) to ( 2 ^ 15 - 1 )
//
// Inputs:
//            no => largest possible value of rand number
//
// Outputs:
//     processor => random processor number, range(1 to no)
int functs::random_funct(int no){
  int x = -1;

  while( x <= 0 || x > no){
    x = (int)((float)no * rand() / RAND_MAX);
  }
  return x;
}; // random_funct

/*******************************************************/
/*                                                     */
/* string2int                                          */
/*                                                     */
/*******************************************************/
//
// This function converts character strings to integers.
//
int functs::string2int(char* word){
  int i, int_val, result;

  result = i = 0;
  for (i=0; word[i] >= '0' && word[i] <= '9'; i++){
    int_val = word[i] - '0';
    result = result*10 + int_val;
  } // while
  return result;
}; // string2int
```

10.9. This executable shell script, `script_java`, is used to compile the Java code for the Counter Server:

```
#!/bin/sh
echo
echo "*********************************"
echo "*  Welcome to the Java script    *"
echo "*********************************"
#    $echo "unix"
CLASSPATH="/usr1/JOB3.1.3/naming/demo/classes:$CLASSPAT
CLASSPATH="/usr1/JOB-3.1.3/naming/lib:$CLASSPATH"
CLASSPATH="/usr1/OB3.1.3/ob/lib/OBNaming.jar:$CLASSPATH"
```

```
CLASSPATH="/usr1/OB-3.1.3/ob/lib/OB.jar:$CLASSPATH"
CLASSPATH="$OB_LIB:$CLASSPATH"
export CLASSPATH

echo "compiling …"
javac ./COUNTER_MODULE/*.java
```

[1] Petrie,C. J., "Agent-Based Engineering, the Web, and Intelligence", *IEEE Expert*, December, 1996.
[2] Genesereth, M. R., "An Agent-Based Approach to Software Interoperability", *Proceedings of the DARPA Software Technology Conference*, 1992.
[3] Schmidt, D. C. and S. Vinoski, "Object Interconnections, Introduction to Distributed Object Computing (Column 1)", *C++ Report Magazine*, January, 1995.
[4] Franklin, S. and A. Graesser, "Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents", *Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages*, Springer-Verlag, 1996.
[5] http://www.crystaliz.com/logicware/mubot.html
[6] Russell, S. J. and P. Norvig, *Artificial Intelligence: A Modern Approach*, Prentice Hall, pp. 33, 1995.
[7] Maes, P., Artificial Life Meets Entertainment: Life like Autonomous Agents", *Communications of the ACM*, Vol. 38, pp. 108-114, 1995.
[8] Smith, D. C., A. Cypher, and J. Spohrer, "KidSim: Programming Agents Without a Programming Language", *Communications of the ACM*, Vol. 37, pp. 55-67, 1994.
[9] Hayes-Roth, B., "An Architecture for Adaptive Intelligent Systems", *Artificial Intelligence*, Vol. 72, pp. 329-365, 1995.
[10] http://activist.gpl.ibm.com:81/WhitePaper/ptc2.htm
[11] Wooldrige, M. and N. R. Jennings, "Agent Theories, Architectures, and Languages: a Survey", *Intelligent Agents*, Springer-Verlag, pp. 1-22, 1995.
[12] http://www.ai.mit.edu/people/sodabot/slideshow/total/P001.html
[13] Brustoloni, J. C., "Autonomous Agents: Characterization and Requirements", Carnegie Mellon Technical Report CMU-CS-91-204, Carnegie Mellon University, 1991.
[14] http://www.ooc.com
[15] Otte, R., P. Patrick, and M. Roy, "Understanding CORBA", Prentice Hall, 1996.
[16] Download OMG Document 98-02-33 from http://www.ooc.com/ob/corba.html